

Управление доступом Обзор



Авторские права

© Postgres Professional, 2017–2024

Авторы: Егор Рогов, Павел Лузанов, Илья Баштанов, Игорь Гнатюк

Фото: Олег Бартунов (монастырь Пху и пик Бхрикути, Непал)

Использование материалов курса

Некоммерческое использование материалов курса (презентации, демонстрации) разрешается без ограничений. Коммерческое использование возможно только с письменного разрешения компании Postgres Professional. Запрещается внесение изменений в материалы курса.

Обратная связь

Отзывы, замечания и предложения направляйте по адресу:

edu@postgrespro.ru

Отказ от ответственности

Компания Postgres Professional не несет никакой ответственности за любые повреждения и убытки, включая потерю дохода, нанесенные прямым или косвенным, специальным или случайным использованием материалов курса. Компания Postgres Professional не предоставляет каких-либо гарантий на материалы курса. Материалы курса предоставляются на основе принципа «как есть» и компания Postgres Professional не обязана предоставлять сопровождение, поддержку, обновления, расширения и изменения.

Роли и атрибуты

Подключение к серверу

Парольная аутентификация

Привилегии и управление ими

Категории ролей

Групповые и предопределенные роли

Привилегии по умолчанию

Привилегии и подпрограммы

Роль может быть пользователем СУБД

не связана с пользователем ОС

Роли можно включать друг в друга

удобно при настройке доступа

Свойства роли определяются атрибутами

LOGIN	возможность подключения
SUPERUSER	суперпользователь
CREATEDB	возможность создавать базы данных
CREATEROLE	возможность создавать роли
и другие	

Роли в PostgreSQL используют для двух целей. Во-первых, роль может выступать как пользователь, подключающийся к СУБД. Во-вторых, роли можно включать друг в друга — это удобно при настройке доступа.

Формально роли никак не связаны с пользователями операционной системы, хотя многие программы это предполагают, выбирая значения по умолчанию. Например, `psql`, запущенный от имени пользователя ОС `student`, автоматически использует одноименную роль `student` (если в ключах утилиты не указана какая-либо другая роль).

При создании кластера определяется одна начальная роль, имеющая суперпользовательский доступ (обычно она называется `postgres`). В дальнейшем роли можно создавать, изменять и удалять.

<https://postgrespro.ru/docs/postgresql/16/database-roles>

Роль обладает некоторыми *атрибутами*, определяющими ее общие особенности и права (не связанные с определенными объектами).

Обычно атрибуты имеют два варианта, например, `CREATEDB` (дает право на создание БД) и `NOCREATEDB` (не дает такого права).

Если у роли есть атрибут `LOGIN`, эта роль — пользователь. Если `NOLOGIN` — роль не является пользовательской и не сможет подключиться к серверу; такие роли обычно используют для включения других ролей.

В таблице перечислены лишь некоторые из атрибутов.

<https://postgrespro.ru/docs/postgresql/16/role-attributes>

<https://postgrespro.ru/docs/postgresql/16/sql-createrole>

Роли и атрибуты

Создадим роль для пользователя Алисы. В команде указаны два атрибута.

В этой теме нам важно, от имени какой роли выполняются команды, поэтому имя текущей роли вынесено в приглашение.

```
student=# CREATE ROLE alice LOGIN PASSWORD 'alice';
```

```
CREATE ROLE
```

Список ролей можно узнать командой:

```
student=# \du
```

List of roles	
Role name	Attributes
alice	
postgres	Superuser, Create role, Create DB, Replication, Bypass RLS
student	Superuser

Обратите внимание, что роль student является суперпользователем. Поэтому до сих пор мы не задумывались о разграничении доступа.

Создадим и базу данных:

```
student=# CREATE DATABASE access_overview;
```

```
CREATE DATABASE
```

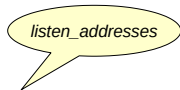
```
student=# \c access_overview
```

You are now connected to database "access_overview" as user "student".

Подключение к серверу

1. Строки `pg_hba.conf` просматриваются сверху вниз
2. Выбирается первая запись, соответствующая параметрам подключения (тип, база, пользователь и адрес)

#	TYPE	DATABASE	USER	ADDRESS	METHOD
	local	all	postgres		peer
	local	all	all		peer
	host	all	all	127.0.0.1/32	scram-sha-256
	host	all	all	:::1/128	scram-sha-256
...					
	local	all — любая роль			
	host	имя роли			
...					
		all — любая БД		all — любой IP	
		имя БД		IP/маска	
				доменное имя	



listen_addresses

5

Для каждого нового клиента сервер определяет, надо ли разрешить подключение к базе данных. Настройки подключения определяются в конфигурационном файле `pg_hba.conf` («host-based authentication»). Как и с основным конфигурационным файлом `postgresql.conf`, изменения вступают в силу только после перечитывания файла сервером (SQL-функцией `pg_reload_conf()` или командой `reload` утилиты управления).

При появлении нового клиента сервер просматривает конфигурационный файл сверху вниз в поисках строки, подходящей к запрашиваемому клиентом подключению. Соответствие определяется по четырем полям: типу подключения, имени БД, имени пользователя и IP-адресу.

Ниже перечислены только самые основные возможности.

Подключение — `local` (unix-сокеты) или `host` (подключение по протоколу TCP/IP).

База данных — ключевое слово `all` (соответствует любой БД) или имя конкретной базы данных.

Пользователь — `all` или имя конкретной роли.

Адрес — `all`, диапазон IP-адресов или доменное имя. Не указывается для типа `local`. По умолчанию PostgreSQL слушает входящие соединения только с `localhost`; обычно параметр `listen_addresses` ставят в значение «*» (слушать все интерфейсы) и дальше регулируют доступ средствами `pg_hba.conf`.

<https://postgrespro.ru/docs/postgresql/16/client-authentication>

3. Выполняется аутентификация указанным методом
4. Удачно — доступ разрешается, иначе — запрещается (если не подошла ни одна запись — доступ запрещается)

#	TYPE	DATABASE	USER	ADDRESS	METHOD
	local	all	postgres		peer
	local	all	all		peer
	host	all	all	127.0.0.1/32	scram-sha-256
	host	all	all	:::1/128	scram-sha-256

trust — разрешить
reject — отказать
scram-sha-256 и md5 — запросить пароль
peer — спросить ОС

Когда в файле найдена подходящая строка, выполняется аутентификация указанным в этой строке методом, а также проверяется наличие атрибута LOGIN и привилегии CONNECT. Если результат проверки успешен, то подключение разрешается, иначе — запрещается (другие строки файла при этом уже не рассматриваются).

Если ни одна из строк не подошла, то доступ также запрещается.

Таким образом, записи в файле должны идти сверху вниз от частных правил к общим.

Существует множество методов аутентификации:

<https://postgrespro.ru/docs/postgresql/16/auth-methods>

Ниже перечислены только самые основные.

Метод trust безусловно разрешает подключение. Если вопросы безопасности не важны, можно указать «все all» и метод trust — тогда будут разрешены все подключения.

Метод reject, наоборот, безусловно запрещает подключение.

Метод scram-sha-256 запрашивает у пользователя пароль и проверяет его соответствие хешу, который хранится в системном каталоге кластера. Парольный метод md5 также используется, но объявлен устаревшим.

Метод peer запрашивает имя пользователя у операционной системы и разрешает подключение, если имя пользователя ОС и пользователя БД совпадают (можно установить и другие соответствия имен).

На сервере

- пароль устанавливается при создании роли или позже
- пользователю без пароля будет отказано в доступе
- пароль хранится в системном каталоге `pg_authid`

Ввод пароля на клиенте

- вручную
- из переменной окружения `PGPASSWORD`
- из файла `~/.pgpass` (строки в формате `узел:порт:база:роль:пароль`)

Если используется аутентификация по паролю, для роли обязательно должен быть установлен пароль, иначе в доступе будет отказано.

Хеш-код пароля хранится в системном каталоге в таблице `pg_authid`.

Пользователь может вводить пароль вручную, а может автоматизировать его ввод. Для этого есть две возможности.

Во-первых, можно задать пароль в переменной окружения `PGPASSWORD` на клиенте. Однако это неудобно, если приходится подключаться к нескольким базам, а также не рекомендуется из соображений безопасности.

Во-вторых, можно задать пароли в файле `~/.pgpass` на клиенте. К файлу должен иметь доступ только его владелец (установлены права доступа 600), иначе PostgreSQL проигнорирует его.

Подключение

Чтобы роль смогла подключиться к базе данных, она должна иметь не только атрибут LOGIN, но и разрешение в файле pg_hba.conf. Располагается он обычно рядом с основным конфигурационным файлом:

```
student=# SHOW hba_file;

          hba_file
-----
/etc/postgresql/16/main/pg_hba.conf
(1 row)
```

А прочитать его содержимое можно прямо из SQL:

```
student=# SELECT type, database, user_name, address, auth_method
FROM pg_hba_file_rules();
```

type	database	user_name	address	auth_method
local	{all}	{all}		trust
host	{all}	{all}	127.0.0.1	scram-sha-256
host	{all}	{all}	:::1	scram-sha-256
local	{replication}	{all}		trust
host	{replication}	{all}	127.0.0.1	scram-sha-256
host	{replication}	{all}	:::1	scram-sha-256

(6 rows)

(В зависимости от сборки сервера содержимое файла может отличаться.)

Мы будем использовать подключение к localhost по TCP/IP (host). Такому подключению соответствует вторая строка выборки. Она предполагает аутентификацию по паролю.

Роль alice была создана с паролем, но вообще его можно изменить в любой момент:

```
student=# ALTER ROLE alice PASSWORD 'alicepass';
```

```
ALTER ROLE
```

Попробуем установить соединение, указав в строке подключения всю необходимую информацию:

```
student$ psql 'host=localhost user=alice dbname=access_overview password=alicepass'
```

```
| alice=> \conninfo
|
| You are connected to database "access_overview" as user "alice" on host "localhost"
| (address "127.0.0.1") at port "5432".
| SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, compression: off)
```

Получилось!

Привилегии определяют права доступа ролей к объектам

Таблицы и представления

SELECT	чтение данных	} можно на уровне столбцов
INSERT	вставка строк	
UPDATE	изменение строк	
REFERENCES	внешний ключ (для таблиц)	
DELETE	удаление строк	
TRUNCATE	опустошение (для таблиц)	
TRIGGER	создание триггеров	

Привилегии устанавливают связь между субъектами (ролями) и объектами кластера. Они определяют действия, доступные для ролей в отношении этих объектов.

Список возможных привилегий отличается для объектов различных типов. Привилегии для основных типов объектов приведены на этом и следующем слайдах.

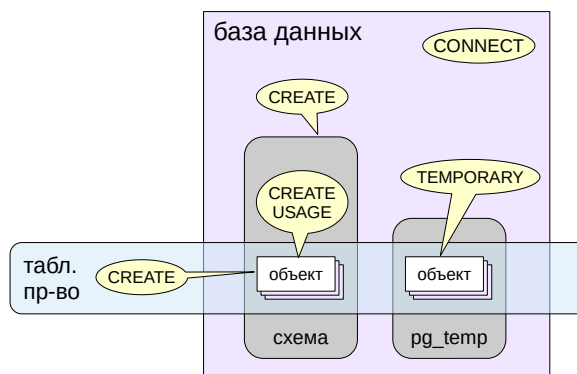
Больше всего привилегий определено для таблиц и представлений. Часть из них можно задать не только для всего отношения, но и для отдельных столбцов.

<https://postgrespro.ru/docs/postgresql/16/ddl-priv>

<https://postgrespro.ru/docs/postgresql/16/sql-grant>

Привилегии

Табличные пространства,
базы данных, схемы



Последовательности

SELECT	currval		
UPDATE		nextval	setval
USAGE	currval	nextval	

10

Возможно, несколько неожиданный набор привилегий имеют последовательности. Выбирая нужные, можно разрешить или запретить доступ к трем управляющим функциям.

Для табличных пространств есть привилегия CREATE, разрешающая создание объектов в этом пространстве.

Для баз данных привилегия CREATE разрешает создавать схемы в этой БД, а для схемы привилегия CREATE разрешает создавать объекты в этой схеме.

Поскольку точное имя схемы для временных объектов заранее неизвестно, привилегия на создание временных таблиц вынесена на уровень БД (TEMPORARY).

Привилегия USAGE схемы разрешает обращаться к объектам в этой схеме.

Привилегия CONNECT базы данных разрешает подключение к этой БД.

Суперпользователи

полный доступ ко всем объектам — проверки не выполняются

Владельцы

изначально все привилегии для объекта (можно отозвать)
действия со своими объектами, не регламентируемые привилегиями,
например: удаление, выдача и отзыв привилегий и т. п.

Остальные роли

доступ в рамках выданных привилегий

В целом можно сказать, что доступ роли к объекту определяется привилегиями. Но имеет смысл выделить три категории ролей и рассмотреть их по отдельности:

1. Роли с атрибутом SUPERUSER (суперпользователи). Такие роли могут делать все, что угодно — для них проверки разграничения доступа не выполняются.
2. Владелец объекта. Изначально это роль, создавшая объект, хотя потом его можно сменить. Владелец считается не только сама роль-владелец, но и любая другая роль, включенная в нее. Владелец объекта сразу получает полный набор привилегий для этого объекта.

В принципе, эти привилегии можно отозвать, но владелец объекта обладает также неотъемлемым правом совершать действия, не регламентируемые привилегиями. В частности, он может выдавать и отзывать привилегии (в том числе и себе самому), удалять объект и т. п.

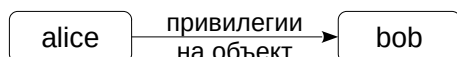
3. Все остальные роли имеют доступ к объекту только в рамках выданных им привилегий.

Чтобы проверить, есть ли у роли необходимая привилегия в отношении некоторого объекта, можно воспользоваться функциями `has_*_privilege`:

<https://postgrespro.ru/docs/postgresql/16/functions-info>

Выдача привилегий

alice: GRANT привилегии ON объект TO *bob*;



Отзыв привилегий

alice: REVOKE привилегии ON объект FROM *bob*;

Право выдачи и отзыва привилегий на объект имеет владелец этого объекта (и суперпользователь).

Синтаксис команд GRANT и REVOKE достаточно сложен и позволяет указывать как отдельные, так и все возможные привилегии; как отдельные объекты, так и группы объектов, входящие в определенные схемы и т. п.

<https://postgrespro.ru/docs/postgresql/16/sql-grant>

<https://postgrespro.ru/docs/postgresql/16/sql-revoke>

Привилегии

Алиса смогла подключиться к базе данных. Теперь она хочет создать для себя схему и несколько объектов в ней.

```
|  alice=> CREATE SCHEMA alice;
|  ERROR:  permission denied for database access_overview
```

В чем проблема?

У Алисы нет привилегии для создания схем в БД. Выдадим ее:

```
student=# GRANT CREATE ON DATABASE access_overview TO alice;
GRANT
```

Пробуем еще раз:

```
|  alice=> CREATE SCHEMA alice;
|  CREATE SCHEMA
```

Теперь, поскольку Алиса является владельцем своей схемы, она имеет все привилегии на нее и может создавать в ней любые объекты. По умолчанию будет использоваться именно эта схема:

```
|  alice=> SELECT current_schemas(false);
|
|  current_schemas
|  -----
|  {alice,public}
|  (1 row)
```

Алиса создает две таблицы.

```
|  alice=> CREATE TABLE t1(n numeric);
|  CREATE TABLE
|  alice=> INSERT INTO t1 VALUES (1);
|  INSERT 0 1
|  alice=> CREATE TABLE t2(n numeric, who text DEFAULT current_user);
|  CREATE TABLE
|  alice=> INSERT INTO t2(n) VALUES (1);
|  INSERT 0 1
```

А суперпользователь создает другую роль для пользователя Боба, который будет обращаться к объектам, принадлежащим Алисе.

```
student=# CREATE ROLE bob LOGIN PASSWORD 'bobpass';
CREATE ROLE
student$ psql 'host=localhost user=bob dbname=access_overview password=bobpass'
```

Боб попытается обратиться к таблице t1.

```
||  bob=> SELECT * FROM alice.t1;
||
||  ERROR:  permission denied for schema alice
||  LINE 1: SELECT * FROM alice.t1;
||                      ^
```

В чем причина ошибки?

Нет доступа к схеме, так как Боб не суперпользователь и не владелец этой схемы.

Проверить права на доступ к схеме можно так (столбец Access privileges):

```
|  alice=> \x \dn+ \x
```

```
Expanded display is on.
List of schemas
-[ RECORD 1 ]-----+-----
Name           | alice
Owner          | alice
Access privileges |
Description    |
-[ RECORD 2 ]-----+-----
Name           | public
Owner          | pg_database_owner
Access privileges | pg_database_owner=UC/pg_database_owner+
               | =U/pg_database_owner
Description    | standard public schema

Expanded display is off.
```

Привилегии отображаются в формате: роль=привилегии/кем_предоставлены.

Каждая привилегия кодируется одним символом, в частности для схем:

- U = usage;
- C = create.

Если имя роли опущено (как в последней строке), то имеется в виду псевдороль public. Обратите внимание, что на схему public у псевдородли public есть лишь привилегия usage. Здесь pg_database_owner соответствует владельцу базы данных.

Если же опущено все поле (как в первой строке), то имеются в виду привилегии по умолчанию: alice имеет обе доступные привилегии на собственную схему, а остальные роли не имеют никаких.

Предоставим доступ к схеме для Боба. Это может сделать Алиса как владелец.

```
| alice=> GRANT CREATE, USAGE ON SCHEMA alice TO bob;
| GRANT
```

Боб снова пробует обратиться к таблице:

```
|| bob=> SELECT * FROM alice.t1;
|| ERROR: permission denied for table t1
```

Снова ошибка. В чем причина?

На этот раз у Боба есть доступ к схеме, но нет доступа к самой таблице. Вот как проверить доступ:

```
| alice=> \dp alice.t1

          Access privileges
Schema | Name | Type | Access privileges | Column privileges | Policies
-----+-----+-----+-----+-----+-----
alice  | t1   | table |                   |                   |
(1 row)
```

И снова видим пустое поле: доступ есть только у владельца, то есть у Алисы.

Алиса предоставляет Бобу доступ на чтение и изменение:

```
| alice=> GRANT SELECT, UPDATE ON alice.t1 TO bob;
| GRANT
```

А для второй таблицы — доступ на вставку и чтение одного столбца:

```
| alice=> GRANT SELECT(n), INSERT ON alice.t2 TO bob;
| GRANT
```

Посмотрим, как изменились привилегии:

```
| alice=> \dp alice.*

          Access privileges
Schema | Name | Type | Access privileges | Column privileges | Policies
-----+-----+-----+-----+-----+-----
alice  | t1   | table | alice=arwdDxt/alice+|                   |
       |      |      | bob=rw/alice        |                   |
alice  | t2   | table | alice=arwdDxt/alice+| n:                 +|
       |      |      | bob=a/alice         | bob=r/alice        |
(2 rows)
```

Теперь пустое поле «проявилось», и мы видим, что в нем находится полный перечень привилегий. Ниже — обозначения,

не все они вполне очевидные:

- a = insert
- r = select
- w = update
- d = delete
- D = truncate
- x = reference
- t = trigger

Привилегии для столбцов отображаются отдельно (столбец Column privileges).

На этот раз попытки Боба увенчаются успехом. Чтобы не указывать каждый раз имя схемы, Боб добавляет его к своему пути поиска.

```
|| bob=> ALTER ROLE bob SET search_path = public, alice;
|| ALTER ROLE
```

Теперь путь поиска будет устанавливаться в каждом сеансе Боба.

```
|| bob=> \c
|| You are now connected to database "access_overview" as user "bob".
|| bob=> SHOW search_path;
||
|| search_path
|| -----
|| public, alice
|| (1 row)
||
|| bob=> UPDATE t1 SET n = n + 1;
|| UPDATE 1
|| bob=> SELECT * FROM t1;
||
|| n
|| ---
|| 2
|| (1 row)
```

Но другие операции по-прежнему запрещены:

```
|| bob=> DELETE FROM t1;
|| ERROR: permission denied for table t1
```

И вторая таблица:

```
|| bob=> INSERT INTO t2(n) VALUES (100);
|| INSERT 0 1
|| bob=> SELECT n FROM t2;
||
|| n
|| ----
|| 1
|| 100
|| (2 rows)
```

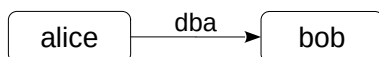
А чтение другого столбца запрещено:

```
|| bob=> SELECT * FROM t2;
|| ERROR: permission denied for table t2
```

Включение роли в роль

Включение в роль

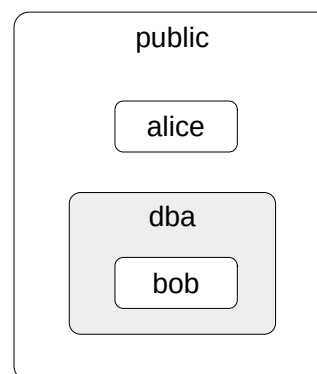
```
alice: GRANT dba TO bob;
```



псевдороль public неявно включает в себя все остальные роли

Исключение из роли

```
alice: REVOKE dba FROM bob;
```



В любую роль могут быть включены другие роли. В таком случае роль выступает в качестве группы. Отдельного понятия «группа» в PostgreSQL нет.

Роль может быть включена в несколько ролей; включенная роль может, в свою очередь, включать в себя третьи роли, но циклы при этом не допускаются.

По умолчанию роль наследует привилегии роли, в которую она включена. Это поведение можно изменить, указав роли атрибут `NOINHERIT` — тогда, чтобы воспользоваться привилегиями включающей роли, надо будет явно переключиться на нее с помощью `SET ROLE`. Атрибуты ролей не наследуются, но можно переключиться на включающую роль и воспользоваться ее атрибутами.

Роли, включающие другие роли, обычно имеют атрибут `NOLOGIN` и называются «групповыми». Фактически это именованный набор привилегий, который удобно «выдать» обычной роли точно так же, как выдается одиночная привилегия. Это упрощает управление доступом и администрирование.

Существует псевдороль `public`, которая неявно включает в себя все остальные роли. Если выдать какие-либо привилегии роли `public`, эти привилегии получают вообще все роли.

<https://postgrespro.ru/docs/postgresql/16/role-membership>

Предопределенные роли

<code>pg_read_all_settings</code>	чтение всех параметров сервера	} <code>pg_monitor</code>
<code>pg_read_all_stats</code>	доступ к статистике	
<code>pg_stat_scan_tables</code>	мониторинг и блокировки таблиц	
<code>pg_read_all_data</code>	чтение данных из всех таблиц	
<code>pg_write_all_data</code>	изменение данных во всех таблицах	
<code>pg_read_server_files</code>	чтение файлов на сервере	
<code>pg_write_server_files</code>	запись в файлы на сервере	
<code>pg_execute_server_programs</code>	выполнение программ на сервере	
...		

В PostgreSQL имеется набор предопределенных ролей, имеющих доступ к часто востребованным, но не общедоступным функциям и данным. Членство в этих ролях может быть предоставлено обычным пользователям для решения ряда задач администрирования — чтобы избежать наделения этих пользователей суперпользовательскими полномочиями.

Количество предопределенных ролей увеличивается с каждой новой версией PostgreSQL. Полный список всех ролей, включая предопределенные, можно просмотреть с помощью команды `\duS` в `psql`.

<https://postgrespro.ru/docs/postgresql/16/predefined-roles>

Можно создавать и свои собственные «административные» роли, например, для выполнения задач резервного копирования.

Групповые роли

Вспомним, что Боб не смог прочитать второй столбец таблицы:

```
|| bob=> SELECT * FROM t2;
|| ERROR: permission denied for table t2
```

Суперпользователь включает Боба в предопределенную роль pg_read_all_data:

```
student=# GRANT pg_read_all_data TO bob;
```

GRANT ROLE

Теперь Боб сможет получить доступ на чтение ко всем таблицам так, как будто ему были выданы привилегии SELECT на все таблицы и USAGE на все схемы:

```
|| bob=> SELECT * FROM t2;
||
||      n | who
||  -----+-----
||      1 | alice
||     100 | bob
||      (2 rows)
```

Получить информацию о членстве в ролях можно с помощью команды \drg утилиты psql:

```
student=# \drg
```

```

              List of role grants
Role name | Member of | Options | Grantor
-----+-----+-----+-----
bob       | pg_read_all_data | INHERIT, SET | postgres
(1 row)
```

Столбец Options содержит информацию об атрибутах, описывающих включение. Здесь для роли bob, включенной в pg_read_all_data, SET означает возможность переключения на групповую роль, а INHERIT — использование привилегий группы без явного переключения на нее.

Исключим пользователя bob из группы:

```
student=# REVOKE pg_read_all_data FROM bob;
```

REVOKE ROLE

Единственная привилегия для функций и процедур

EXECUTE	выполнение
---------	------------

Характеристики безопасности

SECURITY INVOKER	выполняется с правами вызывающего (по умолчанию)
SECURITY DEFINER	выполняется с правами владельца

Для функций и процедур есть единственная привилегия EXECUTE, разрешающая выполнение этой подпрограммы.

Тонкий момент связан с тем, от имени какого пользователя будет выполняться подпрограмма. Если подпрограмма объявлена как SECURITY INVOKER (по умолчанию), она выполняется с правами вызывающего пользователя. В этом случае операторы внутри подпрограммы смогут обращаться только к тем объектам, к которым у вызывающего пользователя есть доступ.

Если же указать фразу SECURITY DEFINER, подпрограмма работает с правами ее владельца. Это способ позволить другим пользователям выполнять определенные действия над объектами, к которым у них нет непосредственного доступа.

<https://postgrespro.ru/docs/postgresql/16/sql-createfunction>

<https://postgrespro.ru/docs/postgresql/16/sql-createprocedure>

Привилегии псевдороль public

подключение к любой базе данных
доступ к системному каталогу
выполнение любых подпрограмм
привилегии выдаются автоматически для каждого нового объекта

Настраиваемые привилегии по умолчанию

возможность дополнительно выдать или отозвать привилегии
для вновь создаваемого объекта

18

Как уже говорилось, псевдороль public включает в себя все остальные роли, которые, таким образом, пользуются всеми привилегиями, выданными для public.

При этом public имеет довольно широкий спектр привилегий по умолчанию. В частности:

- право подключения к любой базе данных (именно поэтому роль alice смогла подключиться к базе данных несмотря на то, что привилегия CONNECT не выдавалась ей явно);
- доступ к системному каталогу;
- выполнение любых подпрограмм.

Это, с одной стороны, позволяет комфортно работать, не задумываясь о привилегиях, а с другой — создает определенные сложности, если разграничение доступа действительно необходимо.

Описанные выше привилегии появляются у public автоматически при создании новых объектов. То есть недостаточно, например, просто отозвать у public привилегию EXECUTE на все подпрограммы: как только появится новая, public немедленно получает привилегию на ее выполнение.

Однако существует специальный механизм «привилегий по умолчанию», который позволяет автоматически выдавать и отзывать привилегии при создании нового объекта. Этот механизм можно использовать и для отзыва права выполнения функций у псевдороль public.

<https://postgrespro.ru/docs/postgresql/16/sql-alterdefaultprivileges>

Подпрограммы и привилегии по умолчанию

Алиса создает функцию:

```
alice=> CREATE FUNCTION foo() RETURNS SETOF t2
AS $$
SELECT * FROM t2;
$$ LANGUAGE sql STABLE;

CREATE FUNCTION
```

Сможет ли Боб вызвать ее, если Алиса не выдала ему привилегию EXECUTE?

```
bob=> SELECT foo();

ERROR: permission denied for table t2
CONTEXT: SQL function "foo" statement 1
```

Вызвать — да, но Боб не сможет получить доступ к объектам, на которые у него нет соответствующих привилегий.

Если Боб создаст свою таблицу t2 в схеме public, функция будет работать для обоих пользователей — но с разными таблицами, поскольку у Алисы и Боба разные пути поиска.

Чтобы Боб мог создать таблицу в схеме public, потребуется явно выдать ему привилегию CREATE на схему (начиная с версии PostgreSQL 15):

```
student=# GRANT CREATE ON SCHEMA public TO bob;
```

GRANT

```
bob=> CREATE TABLE t2(n numeric, who text DEFAULT current_user);

CREATE TABLE

bob=> INSERT INTO t2(n) VALUES (42);

INSERT 0 1

bob=> SELECT foo();

      foo
-----
(42,bob)
(1 row)

alice=> SELECT foo();

      foo
-----
(1,alice)
(100,bob)
(2 rows)
```

Другой доступный вариант — объявить функцию как работающую с правами ее владельца:

```
alice=> ALTER FUNCTION foo() SECURITY DEFINER;

ALTER FUNCTION
```

В этом случае функция всегда будет выполняться с правами Алисы, независимо от того, кто ее вызывает.

Боб удаляет свою таблицу...

```
bob=> DROP TABLE t2;

DROP TABLE
```

...и получает доступ к содержимому таблицы Алисы:

```
bob=> SELECT foo();

      foo
-----
(1,alice)
(100,bob)
(2 rows)
```

В этом случае Алисе надо внимательно следить за выданными привилегиями. Скорее всего, потребуется отозвать

EXECUTE у роли public и выдавать ее явно только нужным ролям.

```
|  alice=> REVOKE EXECUTE ON ALL ROUTINES IN SCHEMA alice FROM public;
|  REVOKE
||
||  bob=> SELECT foo();
||
||  ERROR:  permission denied for function foo
```

Но дело осложняется тем, что по умолчанию привилегия на выполнение автоматически выдается роли public на каждую вновь создаваемую функцию.

Для того чтобы конкретные пользователи получали или, наоборот, лишались тех или иных привилегий на вновь создаваемые объекты, можно настроить привилегии по умолчанию:

```
|  alice=> ALTER DEFAULT PRIVILEGES
|  FOR ROLE alice
|  REVOKE EXECUTE ON ROUTINES FROM public;
|
|  ALTER DEFAULT PRIVILEGES
|
|  alice=> ALTER DEFAULT PRIVILEGES
|  FOR ROLE alice
|  GRANT EXECUTE ON ROUTINES TO bob;
|
|  ALTER DEFAULT PRIVILEGES
|
|  alice=> \ddp
|
|          Default access privileges
|  Owner | Schema |   Type   | Access privileges
|-----+-----+-----+-----+
|  alice |        | function | alice=X/alice   +
|        |        |          | bob=X/alice
| (1 row)
```

Теперь Боб сразу получает привилегию на выполнение подпрограмм, создаваемых Алисой, а остальные пользователи не смогут их выполнять.

```
|  alice=> CREATE FUNCTION bar() RETURNS integer
|  LANGUAGE sql IMMUTABLE SECURITY DEFINER
|  RETURN 1;
|
|  CREATE FUNCTION
|
||  bob=> SELECT bar();
||
||  bar
||  ----
||  1
|| (1 row)
```

Роли, атрибуты и привилегии — гибкий механизм,
позволяющий по-разному организовать работу

- можно легко разрешить все всем

- можно строго разграничить доступ, если это необходимо

При создании новых ролей надо позаботиться
о возможности их подключения к серверу



1. Создайте две роли (пароль должен совпадать с именем):
 - employee — сотрудник магазина,
 - buyer — покупатель.

Убедитесь, что созданные роли могут подключиться к БД.
2. Отзовите у роли public права выполнения всех функций и подключения к БД.
3. Разграничьте доступ таким образом, чтобы:
 - сотрудник мог только заказывать книги, а также добавлять авторов и книги,
 - покупатель мог только приобретать книги.

Проверьте выполненные настройки в приложении.

1. Сотрудник — внутренний пользователь приложения, аутентификация выполняется на уровне СУБД.

Покупатель — внешний пользователь. В реальном интернет-магазине управление такими пользователями ложится на приложение, а все запросы поступают в СУБД от одной «обобщенной» роли (buyer). Идентификатор конкретного покупателя может передаваться как параметр (но в нашем приложении мы этого не делаем).

3. Вообще говоря, разграничение доступа должно быть заложено, в том числе, в приложение. В нашем учебном приложении разграничение не сделано специально: вместо этого на веб-странице можно явно выбрать роль, от имени которой пойдет запрос в СУБД. Это позволяет посмотреть, как поведет себя серверная часть при некорректной работе приложения.

Итак, пользователям нужно выдать:

- Право подключения к БД bookstore и доступ к схеме bookstore.
- Доступ к представлениям, к которым происходит непосредственное обращение.
- Доступ к функциям, которые вызываются как часть API. Если оставить функции SECURITY INVOKER, придется выдавать доступ и ко всем «нижележащим» объектам (таблицам, другим функциям). Однако удобнее просто объявить API-функции как SECURITY DEFINER.

Разумеется, ролям нужно выдать привилегии только на те объекты, доступ к которым у них должен быть.

1. Создание ролей

```
=> CREATE ROLE employee LOGIN PASSWORD 'employee';
```

CREATE ROLE

```
=> CREATE ROLE buyer LOGIN PASSWORD 'buyer';
```

CREATE ROLE

Настройки по умолчанию разрешают подключение с локального адреса по паролю. Нас это устраивает.

2. Привилегии public

У роли public надо отозвать лишние привилегии.

```
=> REVOKE EXECUTE ON ALL FUNCTIONS IN SCHEMA bookstore FROM public;
```

REVOKE

```
=> REVOKE CONNECT ON DATABASE bookstore FROM public;
```

REVOKE

3. Разграничение доступа

Функции с правами создавшего.

```
=> ALTER FUNCTION get_catalog(text,text,boolean) SECURITY DEFINER;
```

ALTER FUNCTION

```
=> ALTER FUNCTION update_catalog() SECURITY DEFINER;
```

ALTER FUNCTION

```
=> ALTER FUNCTION add_author(text,text,text) SECURITY DEFINER;
```

ALTER FUNCTION

```
=> ALTER FUNCTION add_book(text,integer[]) SECURITY DEFINER;
```

ALTER FUNCTION

```
=> ALTER FUNCTION buy_book(integer) SECURITY DEFINER;
```

ALTER FUNCTION

```
=> ALTER FUNCTION book_name(integer,text,integer) SECURITY DEFINER;
```

ALTER FUNCTION

```
=> ALTER FUNCTION authors(books) SECURITY DEFINER;
```

ALTER FUNCTION

Привилегии покупателя: покупатель должен иметь доступ к поиску книг и их покупке.

```
=> GRANT CONNECT ON DATABASE bookstore TO buyer;
```

GRANT

```
=> GRANT USAGE ON SCHEMA bookstore TO buyer;
```

GRANT

```
=> GRANT EXECUTE ON FUNCTION get_catalog(text,text,boolean) TO buyer;
```

GRANT

```
=> GRANT EXECUTE ON FUNCTION buy_book(integer) TO buyer;
```

GRANT

Привилегии сотрудника: сотрудник должен иметь доступ к просмотру и добавлению книг и авторов, а также к каталогу для заказа книг.

```
=> GRANT CONNECT ON DATABASE bookstore TO employee;
```

GRANT

```
=> GRANT USAGE ON SCHEMA bookstore TO employee;

GRANT

=> GRANT SELECT,UPDATE(onhand_qty) ON catalog_v TO employee;

GRANT

=> GRANT SELECT ON authors_v TO employee;

GRANT

=> GRANT SELECT ON operations_v TO employee;

GRANT

=> GRANT EXECUTE ON FUNCTION book_name(integer,text,integer) TO employee;

GRANT

=> GRANT EXECUTE ON FUNCTION authors(books) TO employee;

GRANT

=> GRANT EXECUTE ON FUNCTION author_name(text,text,text) TO employee;

GRANT

=> GRANT EXECUTE ON FUNCTION add_book(text,integer[]) TO employee;

GRANT

=> GRANT EXECUTE ON FUNCTION add_author(text,text,text) TO employee;

GRANT
```

1. Зарегистрируйте пользовательские роли `alice` и `bob`.
2. Отредактируйте файл `pg_hba.conf` так, чтобы беспарольный вход был разрешен лишь для пользователей `postgres` и `student`. Убедитесь, что доступ для `alice` и `bob` запрещен.
3. Для пользователей `alice` и `bob` включите разрешение входить в сеанс, используя метод `peer`. Проверьте невозможность входа в сеанс без создания отображения на пользователя ОС. Для `alice` создайте такое отображение.
4. Проверьте возможность использовать одно и то же отображение для разных ролей.

2. С помощью текстового редактора вставьте перед первой незакомментированной директивой в `pg_hba.conf`

```
local all postgres,student trust
```

Перечитайте конфигурацию.

3. Отредактируйте вставленную в `pg_hba.conf` директиву, заменив ее следующим:

```
local all postgres,student trust
local all alice,bob peer
```

В конец файла `pg_ident.conf` добавьте:

```
stmap student alice
```

4. Чтобы роли `alice` и `bob` использовали одно отображение, содержимое `pg_ident.conf` должно быть следующим:

```
stmap student alice
stmap student bob
```

1. Добавление ролей

Зарегистрируем роли с правом входа в сеанс.

```
=> CREATE ROLE alice LOGIN;
```

```
CREATE ROLE
```

```
=> CREATE ROLE bob LOGIN;
```

```
CREATE ROLE
```

2. Ограничение использования trust

Отредактируем содержимое pg_hba.conf, разрешив метод trust лишь для postgres и student.

```
student$ sudo sed -i 's/^local.*all.*all.*trust.*$/local all postgres,student trust\n/'  
/etc/postgresql/16/main/pg_hba.conf
```

Вот что получилось:

```
=> SELECT type,database,user_name,address,auth_method,error  
FROM pg_hba_file_rules  
ORDER BY rule_number;
```

type	database	user_name	address	auth_method	error
local	{all}	{postgres,student}		trust	
host	{all}	{all}	127.0.0.1	scram-sha-256	
host	{all}	{all}	:::1	scram-sha-256	
local	{replication}	{all}		trust	
host	{replication}	{all}	127.0.0.1	scram-sha-256	
host	{replication}	{all}	:::1	scram-sha-256	

(6 rows)

Применим конфигурацию.

```
=> SELECT pg_reload_conf();
```

```
pg_reload_conf  
-----  
t  
(1 row)
```

Теперь alice и bob не могут подключиться:

```
student$ psql -l -U alice
```

```
psql: error: connection to server on socket "/var/run/postgresql/.s.PGSQL.5432" failed:  
FATAL: no pg_hba.conf entry for host "[local]", user "alice", database "postgres", no  
encryption
```

```
student$ psql -l -U bob
```

```
psql: error: connection to server on socket "/var/run/postgresql/.s.PGSQL.5432" failed:  
FATAL: no pg_hba.conf entry for host "[local]", user "bob", database "postgres", no  
encryption
```

3. Метод аутентификации peer

Используя текстовый редактор, добавим еще одну строку с методом аутентификации peer, чтобы разрешить подключение пользователям alice и bob.

```
student$ sudo sed -i '/^local.*all.*postgres,student.*$/a local all alice,bob peer'  
/etc/postgresql/16/main/pg_hba.conf
```

Содержимое pg_hba.conf:

```
=> SELECT type,database,user_name,address,auth_method,error  
FROM pg_hba_file_rules  
ORDER BY rule_number;
```

type	database	user_name	address	auth_method	error
local	{all}	{postgres,student}		trust	
local	{all}	{alice,bob}		peer	
host	{all}	{all}	127.0.0.1	scram-sha-256	
host	{all}	{all}	::1	scram-sha-256	
local	{replication}	{all}		trust	
host	{replication}	{all}	127.0.0.1	scram-sha-256	
host	{replication}	{all}	::1	scram-sha-256	

(7 rows)

Перечитаем конфигурацию.

```
=> SELECT pg_reload_conf();

pg_reload_conf
-----
t
(1 row)
```

Однако по-прежнему эти учетные записи не могут войти в сеанс, хотя сообщение об ошибке изменилось.

```
student$ psql -l -U alice
```

```
psql: error: connection to server on socket "/var/run/postgresql/.s.PGSQL.5432" failed:
FATAL: Peer authentication failed for user "alice"
```

```
student$ psql -l -U bob
```

```
psql: error: connection to server on socket "/var/run/postgresql/.s.PGSQL.5432" failed:
FATAL: Peer authentication failed for user "bob"
```

Метод peer требует совпадения учетной записи в ОС с именем роли в PostgreSQL. Создадим отображение роли alice на пользователя ОС student, добавив строку в файл pg_ident.conf. Для роли bob отображение задавать не будем.

```
student$ echo 'stmap student alice' | sudo tee -a /etc/postgresql/16/main/pg_ident.conf
```

```
stmap student alice
```

И допишем в добавленную строку параметр map, задающий имя отображения.

```
student$ sudo sed -i 's/peer.*/peer map=stmap/' /etc/postgresql/16/main/pg_hba.conf
```

Содержимое pg_hba.conf:

```
=> SELECT type,database,user_name,address,auth_method,options,error
FROM pg_hba_file_rules
ORDER BY rule_number;
```

type	database	user_name	address	auth_method	options	error
local	{all}	{postgres,student}		trust		
local	{all}	{alice,bob}		peer	{map=stmap}	
host	{all}	{all}	127.0.0.1	scram-sha-256		
host	{all}	{all}	::1	scram-sha-256		
local	{replication}	{all}		trust		
host	{replication}	{all}	127.0.0.1	scram-sha-256		
host	{replication}	{all}	::1	scram-sha-256		

(7 rows)

Перечитаем конфигурацию.

```
=> SELECT pg_reload_conf();

pg_reload_conf
-----
t
(1 row)
```

Теперь alice может подключиться к базе данных и выполнить команды.

```
student$ psql -c '\conninfo' -U alice -d student
```

```
You are connected to database "student" as user "alice" via socket in
"/var/run/postgresql" at port "5432".
```

А bob — нет.

```
student$ psql -c '\conninfo' -U bob -d student
```

```
psql: error: connection to server on socket "/var/run/postgresql/.s.PGSQL.5432" failed:  
FATAL: Peer authentication failed for user "bob"
```

4. Одно отображение для нескольких ролей

Разрешим пользователю bob входить в сеанс на таких же условиях, как alice.

```
student$ echo 'stmap student bob' | sudo tee -a /etc/postgresql/16/main/pg_ident.conf
```

```
stmap student bob
```

Добавленные строки в pg_ident.conf:

```
student$ sudo tail -n2 /etc/postgresql/16/main/pg_ident.conf
```

```
stmap student alice  
stmap student bob
```

Перечитаем конфигурацию.

```
=> SELECT pg_reload_conf();
```

```
pg_reload_conf  
-----  
t  
(1 row)
```

Теперь и alice, и bob смогут подключиться.

```
student$ psql -c '\conninfo' -U alice -d student
```

```
You are connected to database "student" as user "alice" via socket in  
"/var/run/postgresql" at port "5432".
```

```
student$ psql -c '\conninfo' -U bob -d student
```

```
You are connected to database "student" as user "bob" via socket in "/var/run/postgresql"  
at port "5432".
```